Re-Use Dynamic Programming for Sequence Alignment: An Algorithmic Toolkit *

Maxime Crochemore † Gad M. Landau ‡ Baruch Schieber § Michal Ziv-Ukelson ¶

Abstract

The problem of comparing two sequences S and T to determine their similarity is one of the fundamental problems in pattern matching. In this manuscript we will be primarily concerned with sequences as our objects and with various string comparison metrics. Our goal is to survey a methodology for utilizing repetitions in sequences in order to speed up the comparison process. Within this framework we consider various methods of parsing the sequences in order to frame their repetitions, and present a toolkit of various solutions whose time complexity depends both on the chosen parsing method as well as on the string-comparison metric used for the alignment.

1 Preliminaries

Sequence alignment is traditionally based on the transformation of one sequence into the other via operations of substitutions, insertions, and deletions (indels). The costs of the operations are specified in a given scoring matrix δ. An optimal alignment is an alignment that yields the best *similarity score* - a value computed as the sum of the costs of the operations applied in the transformation. The alignment of two sequences A and B can classically be solved in $O(n^2)$ time [43, 57, 61] and $O(n)$ space [29] by dynamic programming. The dynamic programming solution to the string comparison computation problem can be represented in terms of a weighted alignment graph [28]. (See Figure 3. Note that all figures can be found in the Appendix.) An alignment graph for A and B is a directed, acyclic, weighted graph containing $(|A|+1)(|B|+1)$ nodes, each labelled with a distinct pair (x, w) $(0 \leq x \leq |A|,\ 0 \leq w \leq |B|)$. The nodes are organized in a

*The results of this survey were presented at three of the Nato Algorithm Workshops (London String Days 2000, Rouen Sequence Algorithmics 2002, and London Stringology Day 2003).

†Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée Cedex 2, France, `http://www-igm.univ-mlv.fr/~mac/`; Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK; e-mail: `mac@univ-mlv.fr`; partially supported by CNRS, NATO Science Programme, and Wellcome Trust Foundation.

‡Department of Computer Science, Haifa University, Haifa 31905, Israel, Phone: (972-4) 824-0103, Fax: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840; email: `landau@poly.edu`; partially supported by NSF grant CCR-0104307, by the Israel Science Foundation grant 282/01, by the FIRST Foundation of the Israel Academy of Science and Humanities, by IBM Faculty Partnership Award, and by the NATO Science Programme.

§IBM T.J. Watson Research Center, P.O. Box 218 Yorktown Heights, NY 10598; Tel: (914) 945-1169; Fax: (914) 945-3434; e-mail: `sbar@watson.ibm.com`;

¶Department of Computer Science, Technion-Israel Institute of Technology, Technion City, Haifa 32000, Israel, Phone: (972-4) 829-4883 ; email: `michalz@cs.technion.ac.il`; partially supported by the Aly Kaufman Post Doctoral Fellowship, by the Bar-Nir Bergreen Software Technology Center of Excellence, and by the NATO Science Programme.

matrix of $(|A|+1)$ rows and $(|B|+1)$ columns. The alignment graph contains a directed edge with a weight of $\delta(-, b_{w+1})$ from each node (x, w) to $(x, w+1)$, and a weight of $\delta(a_{x+1}, -)$ from each node (x, w) to $(x+1, w)$. Node (x, w) has also a diagonal edge with a weight of $\delta(a_{x+1}, b_{w+1})$ to node $(x+1, w+1)$. The weight of a given edge can be specified directly on the grid graph, or as is frequently the case in biological applications, is given by a scoring matrix which specifies the substitution cost for each pair of characters and the deletion cost for each character from the alphabet. Best-scoring paths in the alignment graph represent optimal alignments of A and B. The dynamic programming algorithm will set the value of vertex (x, w) in the graph to the total weight of the highest scoring path which originates in vertex $[0, 0]$ and ends in vertex $[x, w]$, a weight which is identical to the alignment score between the first x characters of A and the first w characters of B.

The two widely used classes of scoring schemes are distance scoring, in which the objective is to minimize the total alignment score, and similarity scoring, in which the objective is to maximize the total alignment score. Within these classes, scoring schemes are further characterized by the treatment of gap costs. A *gap* is the result of the deletion of one or more consecutive characters in one of the sequences. *Additive* gap costs assign a constant weight to each of the consecutive characters. For other gap functions which have been found useful for biological sequences, see [28].

Note that in sections 3.1 and 3.2 we will formalize the alignment problems at hand in terms of distance minimization. (For the sake of simplicity, we will restrict our examples to the Edit Distance measure [43].) In this respect the first and second subsections differ in style from subsection 3.3 and section 4, where the alignment problems will be formalized in terms of similarity maximization. However, the distance and similarity perspectives are complementary, and any distance problem can be translated into a similarity problem. Correspondingly, the distance-based solutions in this manuscript can easily be translated to score maximization solutions, in order to apply to string comparison metrics which measure similarity, rather than distance. All solutions described in this manuscript assume a scoring scheme with additive gap costs.

2 A High-Level Overview of the Toolkit

Our goal is to develop a methodology for utilizing repetitions in sequences in order to speed up the comparison process. Within this framework we consider various methods of parsing the sequences in order to frame their repetitions (see Figures 5 to 7).

2.1 Stage One: Utilizing Repetitions in the Source Strings.

In Section 3 of this manuscript we present solutions to the *Common Substring Alignment Problem*, (CSA for short), which is defined as follows. We are given a set of one or more strings $S_1, S_2 \ldots S_c$ and a target string T and Y is a common substring of all source strings S_i, that is $S_i = B_i Y F_i$, and Y may be repeated several times in any of the source strings (see Figures 1, 3 and 5). The goal is to compute the similarity of all strings S_i with T, without computing the part of Y over and over again.

More generally, the common substring Y could be shared by different source strings competing over similarity with a common target, or could appear repeatedly in the same source string. Also, in a given application, we could of course be dealing with more than one repeated or shared subcomponent. It is assumed that the locations of the common subsequence Y in each source sequence

S_i are known. However, the part of the target T with which Y aligns, may vary according to each B_i and F_i combination.

We refer the interested reader to [40] for a description of various applications which can be cast as CSA problems. In this manuscript, we will survey the CSA algorithms as macros in a general toolkit for utilizing various forms of repetitions in the graphs used for aligning two sequences.

Let n be the size of the compared sequences, Y denote the common substring, and let ℓ denote the size of Y. Using the classical dynamic programming tables, each appearance of Y in a source string would require the computation of all the values in a dynamic programming table of size $O(n\ell)$.

The three solutions surveyed here are each composed of an *encoding stage* and an *alignment stage*. During the first stage, a data structure is constructed which encodes the comparison of Y with T. Then, during the alignment stage, for each comparison of a source S_i with T, the pre-compiled data structure is used to speed up the part of Y.

A clear distinction should be made between the *off-line* pre-processing work and the *online* encoding stage. In some of the applications for which the CSA algorithms are intended, the source sequence database is prepared *off-line*, while the target can be viewed as an "unknown" sequence which is received *online*. The source strings can be pre-processed *off-line* and parsed into their optimal common substring representation.(The CSA algorithms of Stage One assume that we arrive to the problem after the strings have already been parsed, as opposed to the Stage Two algorithms which will handle the parsing job as well.) Therefore, the CSA algorithms know well beforehand where, in each S_i, Y begins and ends. However, the comparison of Y and T cannot be computed until the target is received. Therefore, the encoding stage, as well as the alignment stage - are both *online* stages, and the tradeoff between the two must be cleverly minimized in order to maximize the efficiency gain by the suggested two-stage scenario. Note that even though both stages are *online*, they do not bear an equal weight on the time complexity of the algorithm. The efficiency gain is based on the fact that the first stage is executed only once per target, and then the encoding results are used, during the second stage, to speed up the alignment of each appearance of the common subcomponent in any of the source strings. In Section 3 the solutions will be analyzed in terms of encoding stage complexity and alignment stage complexity for a single common substring Y. Clearly, the additional running time depends only on the length of the parts of the strings that are not in any common substring.

The results presented in Section 3 include the following three solutions to the Common Substring Alignment Problem, which appear in [40] and [41].

1. Algorithm CSA1: the basic CSA algorithm for unrestricted scoring matrices.

This solution, which is presented in Section 3.1, reduces the $O(n\ell)$ alignment work, for each appearance of the common substring Y in a source string, to $O(n)$ - at the cost of an $O(n^2)$ time encoding stage, which is executed only once. This algorithm is general enough to support scoring schemes which use a scoring matrix whose values are real numbers.

2. Algorithm CSA2: a more efficient CSA algorithm for discrete scoring schemes.

This solution, which described in Section 3.2, consists of an $O(n)$ work alignment stage, which is run for each appearance of the common substring Y in a source string, at the cost of $O(n\ell)$ encoding work, which is executed only once. This algorithm is intended for all scoring schemes which use a scoring matrix whose values are rational numbers.

3. Algorithm CSA3: an even more efficient CSA algorithm for the LCS metric.

Let L_y be the length of the LCS of T and Y, denoted $|LCS[T,Y]|$, and L be $\max\{|LCS[T,S_i]|\}$. This

solution consists of an $O(nL_y)$ time encoding stage that is executed once per common substring, and an $O(L)$ time alignment stage that is executed once for each appearance of the common substring in each source string. This algorithm, which is given in Section 3.3, is intended for the LCS scoring scheme.

2.2 Stage Two: Utilizing Repetitions in Both the Source Strings and the Target Strings.

In Section 4 of the manuscript we further advance by utilizing repetitions in both the source and the target strings. Here, two parsing methods, which are associated with known string compression algorithms, are used for defining the repetitions in the compared sequences. One is the parsing associated with Lempel-Ziv Text Compression, and the other is Run-Length Encoding. Here, the compression parsing is employed, not for the traditional objective of saving storage space or bandwidth, but rather in order to speed up the alignment process. The advantage of compressing both strings is that now the two stages (encoding and alignment) can be interleaved, as the blocks of the compression-partitioned dynamic programming graph are traversed in left-to-right, bottom down order and the output border values of each block are computed from the input border values. Due to the incremental properties of the LZ-partitioned blocks and the simplicity of the run-length blocks, the encoding time bottleneck is eliminated, and the solutions suggested in Stage 1 can be extended to efficiently apply to more advanced scoring schemes.

This leads to the following results, which appear in [13] and are surveyed in Section 4.1.

1. A Sub Quadratic String Comparison algorithm for Unrestricted Scoring Matrices.

The classical algorithm for computing the similarity between two sequences, under string comparison metrics which use a scoring matrix of real number values [58, 61], uses a dynamic programming matrix, and compares two strings of size n in $O(n^2)$ time. The only previously known sub-quadratic global alignment string comparison algorithm, by Masek and Paterson [52], is based on the Four Russians paradigm. The "Four Russians" algorithm divides the dynamic programming table into uniform sized ($\log n$ by $\log n$) blocks, and uses table lookup to obtain an $O(n^2/\log n)$ time complexity string comparison algorithm, based on two assumptions. One is that the sequence elements come from a constant alphabet. The other, which they denote the "discreteness" condition, is that the weights (of substitutions and indels) are all rational numbers.

The algorithms described in [13] present a new approach and are better than the above algorithm in two aspects. First, these algorithms are faster for compressible sequences. For such sequences, the complexity of the [13] algorithms is $O(hn^2/\log n)$, where $h \leq 1$ is the entropy of the sequence.

Second, [13] describes an algorithm that is general enough to support scoring schemes with real number weights. For many scoring schemes, the rational number weights supported by Masek and Paterson's algorithm do not suffice. For example, the entries of PAM similarity matrices, as well as BLOSUM evolutionary distance matrices, are defined to be real numbers, computed as log-odds ratios - and therefore could be irrational.

The paper by Masek and Paterson concludes with the following statement: "The most important problem remaining is finding a better algorithm for the finite (in our terms constant) alphabet case without the discreteness condition". In [13], more than twenty years later, this important open question has finally been answered!

These advantages are based on the following facts. First, the new algorithm does not require any pre-computation of lookup-tables, and therefore can afford more flexible weight values. Also, instead

of dividing the dynamic programming matrix into uniform-sized blocks as did Masek and Paterson, the algorithms described in [13] employ a variable-sized block partition, as induced by Lempel-Ziv factorization of both source and target. The common denominator between blocks, maximized by the compression technique, is then re-cycled and used for computing the relevant information for each block in time which is linear with the length of its sides. In this sense, the approach surveyed in section 4.1 can be viewed as another example of speeding up dynamic programming by keeping and computing only a relevant subset of important values, as demonstrated in [19], [20], [40] and [59]. A similar unbalanced strategy has been successfully used for square detection in strings [14] to speed up the original algorithm based on a divide-and-conquer approach [47].

Note that the notion of "acceleration by text-compression" has been recently applied to a related problem - that of *exact string matching* [35], [51], [60]. It should also be mentioned that another related problem - that of exact string matching in compressed text without decoding it, which is often referred to as "compressed pattern matching", has been studied extensively [1], [21] [55]. Along these lines, string search in compressed text was developed for the compression paradigm of LZ78 [45], and its subsequent variant LZW [63], as described in [36], [56]. A more challenging problem is that of "fully compressed" pattern matching when both the pattern and text strings are compressed [26], [27]. For the LZ78-LZW paradigm, compressed matching has been extended and generalized to that of *approximate pattern matching* (finding all occurrences of a short sequence within a long one allowing up to k changes) in [34], [48].

Here, the adaptable sub-quadratic string comparison algorithms presented in [13] will be described in the context of employing algorithms from the CSA toolkit given in Section 3, as follows. Adapting Algorithm CSA1 to work with the LZ-partitioned blocks leads to an $O(n^2/\log n)$ time and space algorithm for an input of constant alphabet size. For most texts, the time and space complexity is actually $O(hn^2/\log n)$ where $h \leq 1$ is the entropy of the text. This algorithm applies to string comparison metrics that utilize a scoring matrix of real value numbers.

Adapting Algorithm CSA2 to the LZ-parsing block partitioning framework leads to a reduction of the space complexity to $O(h^2n^2/(\log n)^2)$, while preserving the $O(hn^2/\log n)$ time complexity. This algorithm applies to string comparison metrics that utilize a scoring matrix of rational value numbers.

Note that, for the sake of presentation simplicity, we assume, throughout the description of the global alignment and the local alignment solutions (subsections 4.2 and 4.3), that both input strings A and B are of the same size n, and that both sequences share the same entropy h. For the case of comparing string A of size m and entropy $0 < h_A \leq 1$ with string B of size n and entropy $0 < h_B \leq 1$, the results of subsections 4.2 and 4.3 are as follows.

- $O(mn(h_A/\log m + h_B/\log n))$ time and space complexity for both global alignment and local alignment, replaces the $O(hn^2/\log n)$ result.
- $O(h_Ah_Bmn/\log m \log n)$ space complexity for global alignment over "discrete" scoring matrices, replaces the $O(h^2n^2/(\log n)^2)$ result.

2. String Comparison of Run Length Compressed Strings.

A string S is *run-length encoded* if it is described as an ordered sequence of pairs (σ, i), often denoted "σ^i," each consisting of an alphabet symbol, σ, and an integer, i. Each pair corresponds to a *run* in S, consisting of i consecutive occurrences of σ. For example, the string *aabbbbbccc* can be encoded as $a^2b^5c^3$. Such a run-length encoded string can be significantly shorter than the expanded string representation after efficiently encoding the integers (see [17] for example). Run-length encoding

serves as a popular image compression technique, since many classes of images (e.g., binary images in facsimile transmission or for use in optical character recognition) typically contain large patches of identically-valued pixels.

Let m and n be the lengths of two run-length encoded strings X and Y, of encoded lengths m' and n', respectively. Previous algorithms for the problem compared two run-length encoded strings using the Levenshtein Edit Distance [43] and the LCS similarity measure [30]. For the LCS metric, Bunke and Csirik [11] presented an $O(mn' + nm')$ time algorithm, while Apostolico, Landau, and Skiena [5] described an $O(m'n' \log(m'n'))$ time algorithm. Mitchell [46] has obtained an $O((d + m' + n') \log(d + m' + n'))$ time algorithm for a more general string matching problem in run-length encoded strings, where d is the number of matches of compressed characters. Both Arbell *et al.* [7] and Mäkinen *et al.* [49] independently obtained an $O(m'n+n'm)$ time algorithm for computing the edit distance between two run-length encoded strings for the Levenshtein distance metric.

Mäkinen, Navarro and Ukkonen [49] posed as an open problem the challenge of extending these results to more general scoring schemes, since in those applications which are related to image compression, the change from a pixel value to the next is smooth. This open problem was answered in [13], where it was shown that adapting Algorithm CSA1 to the block partitioning obtained by run-length encoding yields an $O(m'n + n'm)$ complexity algorithm for comparing two *run-length* encoded strings of length m and n, compressed into m' and n' runs respectively. This result, which is described in section 4.2, extends to all distance or similarity scoring schemes which use an additive gap penalty.

We also observed that additional methods to achieve linear-time propagation across run-length compressed blocks can be obtained by employing algorithm CSA2 or alternatively by adapting the queue algorithms described in [18, 24, 31]. In [50], Mäkinen *et al.* obtained the same bounds for weighted edit distance, using different techniques.

3 Utilizing Repetitions in Source Strings: Common Substring Alignment (CSA) Algorithms.

The alignment graph used for computing the distance between a source string $S_i = B_i Y F_i$ and a target string T can be viewed as a concatenation of three sub-graphs, where the first graph represents the distance between B_i and T, the second graph represents the distance between Y and T, and the third graph represents the distance between F_i and T (see Figure 3). Let ℓ denote the size of the subsequence Y which is common to all S_i.

In this partitioned solution, the weights of the vertices in the last row of the first graph serve as input to initialize the weights of the vertices in the first row of the second graph. The weights of the last row of the second graph can be used to initialize the first row of the third graph. Therefore, let G denote the second sub-graph for comparing Y and T. Let I denote the series of weights of the vertices in the first row of G. Let O denote the series of weights of the vertices of the last row of G. The motivation for breaking the solution into three sub-graphs is that the second sub-graph, representing the distance between Y and T, is identical for all alignment graphs comparing any of the strings S_i with T (see Figure 5). More specifically, both the structure and the weights of the edges of all alignment sub-graphs comparing Y with T are identical, but the weights to be assigned to the vertices during the distance computation may vary according to the prefix B_i which is specific to the source string. Therefore, an initial investment in the learning of this graph as an

encoding stage, and in its representation in a more informative data structure, may pay off later on.

The CSA solutions described in this section comply by the following 2-stage approach.

Encoding Stage: Given Y and T, encode G in a format which can be efficiently used by the alignment stage.

Alignment Stage: Given the output of the encoding stage and input row I - compute the output row O.

During the encoding stage, a data structure is constructed which encodes the comparison of Y with T. Then, during the alignment stage, for each comparison of a source S_i with T, the pre-compiled data structure is used to speed up the part of aligning each appearance of the common substring Y.

3.1 Algorithm CSA1: An $O(n^2)$ Encoding and $O(n)$ Alignment CSA Algorithm for Unrestricted Scoring Matrices

The algorithm described in this subsection consists of an $O(n^2)$ encoding stage and an $O(n)$ alignment stage. Note that the time complexity of comparing a source string of size ℓ and a target string of size n, using the classical dynamic programming algorithm, is $O(n\ell)$. Therefore, there are cases ($n = \Omega(\ell k)$ where k denotes the total number of appearances of the common substring Y in the database) where, due to the encoding stage bottleneck, it is not beneficial or even wasteful to employ this algorithm. In any case, this basic algorithm comes as a stepping stone in preparation for the more efficient two dimensional Common Substring Alignment algorithms, to be presented in section 4. In fact, in section 4 we will show that algorithm CSA1 does very well when both source and target strings are compressed, in which case the alignment stage bottleneck associated with this algorithm is eliminated.

The advantage of Algorithm CSA1 is in the generality of the scoring schemes to which it applies: it supports any additive gap scheme which utilizes a scoring matrix of real values.

The Encoding Stage

The following $DIST$ matrix is computed (see Figure 8).

Definition 1 *$DIST[i, j]$, for $j = 0 \ldots n, i = 0 \ldots j$, stores the weight of the shortest path from the vertex in column i of the first row of the graph G to the vertex in column j of the last row of the graph G.*

A similar encoding of an alignment graph has been used in [4, 8, 10, 33, 59].

A $DIST$ matrix for an $n \times \ell$ sized graph G (encodes the comparison of T versus Y) can be constructed in $O(n^2 + n\ell)$ time by using the algorithm of [59] (note that ℓ could be greater than n). For the LCS and Edit Distance metrics, $DIST$ can also be constructed in $O(n^2 + n\ell)$ time by employing [38]. Apostolico *et al.* [3] show how to construct $DIST$ in $O(n^2)$ time. Alternatively, one could use the algorithm from [4] to construct $DIST$ in $O(n^2 \log n)$ time.

The Alignment Stage

Given input row I and sub-graph G, the weight of output row vertex O_j can be computed as follows.

$$O_j = \min_{r=0}^{j}\{I_r + DIST[r, j]\}$$

This computation entails selecting a minimum among up to n sums for each of the n output sources. (Figure 4 demonstrates an example of an output entry computation.) The above formulation was first presented in [33]. It is, in essence, a static version of the 1D dynamic programming problem [22]:

$$E[j] = \min_{i=0}^{j}\{D[i] + w(i, j)\}$$

in which all values of $D[i]$ are specified before any value of $E[j]$ is computed, and the values of function $w(i, j)$ are precomputed for all integers $j = 0 \ldots n, i = 0 \ldots j$.

Value O_j is the minimum in column j of the following OUT matrix, which merges the information from input row I and $DIST$. (See Figure 8).

Definition 2 *$OUT[i, j] = I_i + DIST[i, j]$ for $j = 0 \ldots n, i = 0 \ldots j$.*

Aggarwal and Park [8] and Schmidt [59] observed that $DIST$ matrices are Monge arrays [54].

Definition 3 *A matrix $M[0 \ldots m, 0 \ldots n]$ is* **Monge** *if either condition 1 or 2 below holds for all $i = 0 \ldots m; j = 0 \ldots n$:*

Condition 1: *$M[a, d] - M[a, c] \geq M[b, d] - M[b, c]$ for all $a < b$ and $c < d$.*

Condition 2: *$M[a, d] - M[a, c] \leq M[b, d] - M[b, c]$ for all $a < b$ and $c < d$.*

We use the crossing paths contradiction [59] to show that our version of $DIST$ obeys Monge Condition 1. Consider the four optimal (lowest scoring) paths from vertices a, b in the first row of G to vertices c, d in the last row of G, as shown in Figure 2.

Path t - corresponds to $DIST[a, c]$.

Path w - corresponds to $DIST[a, d]$.

Path z - corresponds to $DIST[b, c]$.

Path y - corresponds to $DIST[b, d]$.

Note that *Path w* and *Path z* must intersect at some column of the Dynamic Programming Graph before or at column c. Let C denote the weight of the prefix of *Path w* from its origin up to the intersection point, and let D denote the weight of the suffix of *Path w* from the intersection point on. Let A denote the weight of the prefix of *Path z* from its origin up to the intersection point, and B denote the weight of the suffix of *Path z* from the intersection point on. Let $|t|, |w|, |y|$ and $|z|$ denote the weights of paths t, w, y and z, correspondingly.

Since path t is optimal, $|t| \leq C + B \Rightarrow |w| - |t| \geq |w| - C - B = D - B$.

Since path y is optimal, $|y| \leq A + D \Rightarrow |y| - |z| \leq A + D - |z| = D - B$.

Therefore,

$$|w| - |t| = DIST[a, d] - DIST[a, c] \geq DIST[b, d] - DIST[b, c] = |y| - |z|.$$

Since $DIST$ is Monge by Condition 1, so is OUT, which is a $DIST$ with constants added to its rows. An important property of Monge arrays is that of being totally monotone.

Definition 4 *A matrix $M[0\ldots m, 0\ldots n]$ is* **totally monotone** *if either condition 1 or 2 below holds for all $a, b = 0\ldots m;\ c, d = 0\ldots n$:*

1. *$M[a,c] \geq M[b,c] \Longrightarrow M[a,d] \geq M[b,d]$ for all $a < b$ and $c < d$.*
2. *$M[a,c] \leq M[b,c] \Longrightarrow M[a,d] \leq M[b,d]$ for all $a < b$ and $c < d$.*

Note that the Monge property implies total monotonicity, but the converse is not true.

The recursive algorithm of Aggarwal et al [6], nicknamed $SMAWK$ in the literature, can be used to compute in $O(n)$ time all the column minima of an $n \times n$ totally monotone matrix, by querying only $O(n)$ elements of the array. Hence, one could use $SMAWK$ to compute the output row O by querying only $O(n)$ elements of OUT. Clearly, if both the full $DIST$ and all entries of I are available, then accessing an element of OUT is O(1) work.

One obstacle which comes up during this implementation is that $DIST$ is not rectangular. Only the values in the upper triangle are defined. However, this can be resolved by setting the undefined values in the lower triangle to ∞.

Time and Space Complexity Analysis of Algorithm CSA1. $DIST$ is constructed during the encoding stage in $O(n^2)$ time and space. The alignment stage is done in $O(n)$ time and space, for each appearance of the common substring Y in a source sequence.

3.2 Algorithm CSA2: An $O(n\ell)$ Encoding and $O(n)$ Alignment CSA Algorithm for Discrete Scoring Schemes

The second CSA algorithm comes as an answer to the following challenge: can the encoding stage complexity be reduced, without increasing the time complexity of the alignment stage? In order to make the CSA approach beneficial in all cases, our objective is to invest no more than $O(n\ell)$ work in encoding an $n \times \ell$-sized common substring block. We present a more efficient, non-recursive algorithm which applies to all "discrete" scoring schemes, i.e. any additive gap scoring schemes which use a scoring matrix whose entries are rational numbers. Algorithm CSA2 obtains its strength advantage from the fact that it utilizes the Monge property, while the basic Algorithm CSA1 which was described in the previous subsection utilizes only the weaker Total Monotonicity property. This, combined with the discreteness properties of the allowed scoring schemes, yields a reduction of the encoding stage complexity from $O(n^2)$ down to $O(n\ell)$. Since $O(n\ell)$ is the default time complexity for applying classical dynamic programming to any $O(n\ell)$-sized instance on the common sub-graph, this algorithm is a winner under any common substring parsing configuration.

The Encoding Stage

The encoding stage of Algorithm CSA2 gains its efficiency by utilizing the fact that the number of relevant changes, from one column of both OUT and $DIST$ to the next, is constant [39, 40, 59]. This property allows for a representation of $DIST$ via an $O(n)$ number of "relevant" points.

The $DELTA$ matrix is defined as follows.

Definition 5 $DELTA[i,j] = OUT[i,j] - OUT[i,j-1] = DIST[i,j] - DIST[i,j-1]$ *for* $j = 1\ldots n, i = 0\ldots j-1$.

The range of possible values for $DELTA[i,j]$ depends on the scoring scheme which is used for the string comparison, and is actually the upper bound for the value difference between two consecutive elements in the dynamic programming table. (For an example of a $DELTA$ matrix, see Figure 9.)

We will use the term ψ to denote the range bound for $DELTA[i,j]$ values. As an example, if the similarity metric used is LCS, the only possible values for $DELTA$ will be either 1 or 0, and ψ assumes a value of 1. For the Edit Distance metric, on the other hand, ψ is 2, since $DELTA$ can only assume one of the 3 values: -1, 0, 1 [62]. Our algorithm applies to all scoring scheme metrics for which ψ is a constant. It can be shown [52] that ψ is a constant for all scoring schemes which use a scoring matrix whose values are rational numbers.

Note that the following two observations apply to any column in $DELTA$.

Observation 1

Since $DIST$ is a Monge array, each column in $DELTA$ is a series of monotonically non-increasing values.

Observation 2

Since the range of distinct values which $DELTA$ may assume is bounded by a constant (ψ), the number of "steps" (row indices in which the series of column entries increases in value) in each column of $DELTA$ is constant.

As a result, $DIST$ can be represented via an $O(\psi n)$ size set of relevant "step" points collected from all columns of $DELTA$ (see Figure 9).

Definition 6 *Let $Borderline[\alpha, j]$, for $\alpha = 0 \ldots \psi$ and $j = 0 \ldots n$, denote a row index of a "step" of size 1 in the series of monotonically non-increasing values of column j of $DELTA$. (A step of size k is represented by k different Borderline Points).*

Clearly, $DELTA$ has up to ψn Borderline Points.

Observation 3

For any two rows x_1, x_2 where $x_1 \leq Borderline[\alpha, j] < x_2$ for any $\alpha = 0 \ldots \psi$, for some column j of $DELTA$,

$$OUT[x_2, j] - OUT[x_2, j-1] < OUT[x_1, j] - OUT[x_1, j-1]$$

During the encoding stage, the $O(n\ell)$ time complexity algorithm of [[59], section 6] is used to compute the Borderline Points of $DELTA$. For the LCS metric, $DELTA$ can also be constructed in $O(nL_y)$ time, where $L_y = |LCS[T,Y]| \leq \ell$, by employing the algorithms of [39].

The Alignment Stage

We now present an algorithm which uses the pre-compiled *Borderline Points* and input row I to compute output row O. The new algorithm computes the column minima of OUT, in left-to-right order, by executing n iterations. It employs the candidate list concept ([18], [24], [31], [53]), as follows.

Definition 7 *The* **candidate list**, *at iteration j of the algorithm, is a subset of the rows of column j of OUT that includes only those rows which are candidates to contain future OUT column minima.*

The candidate list is updated from one iteration of the alignment stage algorithm to the next. It is sorted by increasing OUT value and increasing row index. At iteration j of the alignment stage

algorithm, the candidate of smallest row index in the list bears the minimal value at column j of OUT.

A high-level outline of the alignment stage algorithm is given below.

Procedure Alignment Stage

input: The set of Borderline Points for the $DELTA$ matrix, and input row I.

output: The output row O.

for $j := 0$ to $|T|$ *do*

1 Append candidate j, which represents to row j of OUT, to the candidate list.

2 Update the contents of the candidate list.

3 Report output entry O_j.

The list contents are updated at each iteration by removing rows which are no longer candidates to produce future column minima. We will denote such rows as *extinct*, according to total monotonicity condition 1.

Definition 8 *A row* x_1 *is* **extinct** *at iteration* j *of the candidate list algorithm, for* $0 < j \leq n$*, if* $\{\exists\ x_2 > x_1 \mid OUT[x_2, j] \leq OUT[x_1, j]\}$.

Hence, for any distinct output value achieved during the computation of O_j, we only need to keep one representative - that of the greatest candidate to achieve this value. All other candidates which generate the same output value in iteration j become extinct.

More specifically, a candidate becomes *extinct*, by Definition 8, as a result of two possible events.

Event 1. A $Bordeline[\alpha, j]$. Let x_1, x_2 denote two rows which appear sequentially on the candidate list at iteration j, where $x_1 \leq Bordeline[\alpha, j] < x_2$. For any two rows x_1, x_2 where $x_1 \leq Borderline[\alpha, j] < x_2$.

$$OUT[x_2, j] - OUT[x_2, j-1] < OUT[x_1, j] - OUT[x_1, j-1]$$

The candidate list is sorted by increasing OUT value and increasing row index. Therefore, by Observation 3, a $Bordeline[\alpha, j]$ could result in x_2 reaching an identical value to x_1 at column j of OUT. The difference in the OUT value between any two active candidates of index $> Bordeline[\alpha, j]$ is not affected by the $Bordeline[\alpha, j]$ event. The difference in the OUT value between any two active candidates of index $\leq Bordeline[\alpha, j]$ is also unaffected by the $Bordeline[\alpha, j]$ event. The only two candidates which may now generate the same value at iteration j, as a direct result of the $Bordeline[\alpha, j]$ event, are the candidate x_1 which is of highest index below $Bordeline[\alpha, j]$, and the candidate x_2 which is of lowest row index above $Bordeline[\alpha, j]$, since the difference in the OUT values between these two candidates decreases by one. As a result, row x_1 will be removed from the list (row 5 in column 8 of the OUT matrix of Figure 10).

Event 2. At iteration j, row j is appended to the list. At this point, j is the highest row index in the list, and therefore all the elements with an OUT value which is higher than or equal to that of candidate j will be removed from the list (row 2 in column 3 and rows 1 and 3 in column 4 of the OUT matrix of Figure 10).

Note that two technical challenges need to be met, in order to implement a list manipulation engine, which updates the contents of the candidate list in linear time.

1. **Computing the OUT value of a candidate.**

   The values of OUT are needed for two purposes. One is the comparison of two adjacent candidates. The other is for reporting the OUT value O_j. Since OUT is quadratic in size, it is not possible to compute the OUT value for each candidate on the list in each column of OUT in linear time. Instead, Algorithm CSA2 keeps track of the differences only in OUT values between the members of the candidate list. Column j of the candidate list is computed incrementally from column $j-1$ of the candidate list by applying the relevant borderline points. The differences in OUT values between two candidates will only change in the positions updated by the borderline points. Since the number of borderline points per column is constant, in [40] we are able to show how to compute all the necessary OUT values in linear time.

2. **Efficiently accessing the rows to be removed from the candidate list.**

   Since not all rows of OUT appear in the candidate list, finding the row to be removed as a result of a Borderline Point is not trivial (see Event 1). However, this can be solved if the candidate list is implemented by employing the *incremental tree set union* algorithm described in [[23], pp. 216], for the special case in which the union tree is a path.

Time and Complexity Analysis of Algorithm CSA2.

In the encoding stage the Borderline Points are computed in $O(n\ell)$ time and $O(n)$ space using [59].

In [40] it is proven that the alignment stage of Algorithm CSA2 computes output row O, from input row I and the Borderline Points for the comparison of Y with T, in $O(n)$ time and space.

Note that during the encoding stage, the borderline points for the comparison of the prefix Y_1^j with T, can be incrementally computed in $O(n)$ time from the borderline points for the comparison of Y_1^{j-1} with T, using [59]. Hence, for source sequences with two or more common factors, the time complexity of the encoding stage is further reduced to $O(nD)$, where D is the number of nodes in the dictionary trie for the common factors.

3.3 Algorithm CSA3: An $O(nL)$ Encoding and $O(L)$ Alignment CSA Algorithm for the LCS metric.

The third CSA algorithm comes as an answer to the following challenge: can a more efficient common substring alignment algorithm, which exploits the *sparsity* inherent to the LCS metric, be designed for the LCS metric?

This algorithm consists of an $O(nL_y)$ encoding stage, and an $O(L)$ alignment stage, where L_y denotes the length of the LCS of T and Y, denoted by $|LCS[T,Y]|$, and L denotes $\max\{|LCS[T,S_i]|\}$. When the problem is sparse ($L_y << |Y|$, $L << n$), the time bounds of Algorithm CSA3 are better than those of Algorithms CSA1 and CSA2. Even when the data is dense, the time bounds of this algorithm are not worse than the previous two algorithms.

Algorithm CSA3 exploits sparsity, based on the following two observations.

1. The LCS I and O vectors are both monotone staircases with at most $L+1$ unit steps (see Figure 11).

2. Each row in the LCS $DIST$ is a monotone staircase with with at most $L+1$ unit steps (see Figure 12).

In [41] it is proven that the $O(L)$ steps of I are sufficient for the computation of the $O(L)$ steps of O. This allows the two vectors to be compressed into their "step" representation.

The Encoding Stage

In [41] we show how the $DIST$ matrix, which is traditionally used to encode the comparison of Y and T, can be replaced with a smaller matrix (see Figure 12), which is in essence the compressed, "step" representation of the rows of the original $DIST$. This "step" compressed $DIST$ can be computed in $O(nL_y)$ time using the Consecutive Suffix Alignments algorithms described in [39].

The Alignment Stage

Algorithm CSA3 propagates the steps of I across the "step" compressed $DIST$ matrix. In [41] it is shown that the OUT matrix which can be obtained from combining the steps of I with the steps of $DIST$ preserves the Total Monotonicity condition, thus enabling an efficient adaptation of the matrix searching algorithm of Aggarwal *et al.* [6]. This adaptation is then used to compute the $O(L)$ steps of O from the $O(L)$ steps of I in $O(L)$ time.

4 Utilizing Repetitions in Source Strings as well as Target Strings.

In this section we expand re-use Dynamic Programming to handle repetitions in both source and target strings. The repetitions are defined via two known parsing algorithms: one is the parsing associated with LZ78 compression. The other is run-length encoding. The results are obtained by adapting and expanding CSA algorithms from the toolkit given in the previous section.

In order to avoid confusion, we remind the reader that throughout this section the alignment problem will be re-formalized in terms of similarity maximization rather than distance minimization. (This, in spite of the fact that in the previous Section 3 algorithms CSA1 and CSA2 were presented using an edit-distance minimization formalism.) Therefore, in both subsections 4.1 and 4.2, we use the term "optimal path" to denote a highest scoring path.

4.1 A Block Partitioning of the Alignment Graph based on LZ78 Parsing

The classical algorithm for computing the similarity between two sequences [58, 61] uses a dynamic programming matrix, and compares two strings of size n in $O(n^2)$ time. (For the sake of presentation simplicity, we assume, throughout this section, that both input strings A and B are of the same size n, and that both sequences share the same entropy h.) The speed-up is achieved by dividing the dynamic programming matrix into variable sized blocks, as induced by Lempel-Ziv parsing of both strings, and utilizing the inherent periodic nature of both strings. The compression mechanism enables the algorithm to adapt to the data and to utilize its repetitions. In this section we will describe an $O(n^2/\log n)$ algorithm for computing the optimal global alignment value of two strings, of size n each, over a constant alphabet. The algorithm is even faster when the sequences are compressible. In fact, for most texts, the complexity of our algorithm is actually $O(hn^2/\log n)$. For global alignment over "discrete" scoring matrices, we explain how the space complexity can be reduced to $O(h^2n^2/(\log n)^2)$, without impairing the $O(hn^2/\log n)$ time complexity.

The LZ compression methods are based on the idea of self reference: while the text file is scanned, substrings or phrases are identified and stored in a dictionary, and whenever, later in the process,

a phrase or concatenation of phrases is encountered again, this is compactly encoded by suitable pointers [42], [44], [45]. Of the several existing versions of the method, we will use the ones which are denoted LZ78 family [63], [45]. The main feature which distinguishes LZ78 factorization from previous LZ compression algorithms is in the choice of codewords. Instead of allowing pointers to reference any string that has appeared previously, the text seen so far is parsed into phrases, where each phrase is the longest matching phrase seen previously plus one character. For example, the string "B = aacgacgat" is divided into five phrases: a, ac, g, acg, at. Each phrase is encoded as an index to its prefix, plus the extra character (see Figure 14). The new phrase is then added to the list of phrases that may be referenced.

Since each phrase is distinct from others, the following upper bound applies to the possible number of phrases obtained by LZ78 factorization.

Theorem 4.1.1 (Lempel and Ziv 1976 [42]) *Given a sequence S of size n over a constant alphabet. The maximal number of distinct phrases in S is $O(\frac{n}{\log n})$.*

Even though the upper bound above applies to any possible sequence over a constant alphabet, it has been shown that in many cases we can do better than that. Intuitively, the LZ78 algorithm compresses the sequence because it is able to discover some repeated patterns. Therefore, in order to compute a tighter upper bound on the number of phrases obtained by LZ78 factorization for "compressible" sequences, the repetitive nature of the sequence needs to be quantified. One of the fundamental ideas in information theory is that of *entropy*, denoted by the real number h, $0 < h \leq 1$, which measures the amount of disorder or randomness, or inversely, the amount of order or redundancy in a sequence. Entropy is small when there is a lot of order (i.e, the sequence is repetitive) and large when there is a lot of disorder. The entropy of a sequence should ideally reflect the ratio between the size of the sequence after it has been compressed, and the length of the uncompressed sequence. The number of distinct phrases obtained by LZ78 factorization has been shown to be $O(hn/\log n)$ for most texts [9], [15], [42], [45]. Note that for any text over a constant alphabet, the upper bound above still applies by setting h to 1.

The alignment graph will be partitioned as follows. Strings A and B will be parsed separately using LZ78 factorization. This induces a partition of the alignment graph for comparing A with B into variable-sized blocks (see Figure 7). Each block will correspond to a comparison of an LZ phrase of A with an LZ phrase of B.

Let xa denote a phrase in A obtained by extending a previous phrase x of A with character a, and yb denote a phrase in B, obtained by extending a previous phrase of B with character b.

From now on we will focus on the computations necessary for a single block of the alignment graph. Consider the block G which corresponds to the comparison of xa and yb. Extending the definitions of section 3.1, we define *input border* I as the left and top borders of G, and *output border* O as the bottom and right borders of G. (The node entries on the input border are numbered in a clockwise direction, and the node entries on the output border are numbered in a counter-clockwise direction.)

Let ℓ_r denote the number of rows in G, $\ell_r = |xa|$. Let ℓ_c denote the number of columns in G, $\ell_c = |yb|$. Let $t = \ell_r + \ell_c$. Clearly, $|I| = |O| = t$.

We define the following three $prefix$ blocks of G.

1. The $left\ prefix$ of G denotes the block comparing phrase xa of A and phrase y of B.
2. The *diagonal* $prefix$ of G denotes the block comparing phrase x of A and phrase y of B.

3. The *top prefix* of G denotes the block comparing phrase x of A and phrase yb of B.

Observation 4

When traversing the blocks of an LZ78 parsed alignment graph in a left-to-right, top-to-bottom order, the blocks for the left prefix, diagonal prefix and top prefix of G are encountered prior to block G (see Figure 7). Note that the graph for the left prefix of G is identical to the subgraph of G containing all columns but the last one. More specifically, both the structure and the weights of edges of these two graphs are identical, but the weights to be assigned to the vertices during the similarity computation may vary according to the input border values. Similarly, for the top prefix and diagonal prefix graphs. The only new cell in G, which does not appear in any of its prefix block graphs, is the cell in the bottom-rightmost corner of G, which is the cell for comparing a and b.

4.2 Combining Algorithm CSA1 with LZ78 Block Partitioning.

Similarly to Algorithm CSA1, the work for each block consists of two stages.

The Encoding Stage

The structure of G is encoded by computing a $DIST$ matrix containing weights of optimal paths connecting each entry of its input border with each entry of its output border. Here we will use the incremental properties of LZ78 parsing to implement the encoding for each block in time linear with the size of its borders. Assuming that a Prefix Block can be accessed in constant time, the new $DIST$ information for G can be computed in $O(t)$ time, by using the $DIST$ representations previously computed for its prefix blocks, plus the information of its new cell, as follows (see Figure 13). Only one new $DIST$ column, column ℓ_c, needs to be computed for the encoding of G, and this new column contains the weights of optimal paths from each input entry to the single new vertex of G, which is vertex (ℓ_r, ℓ_c) in the lowest, rightmost corner of G. All the previous $DIST$ columns are obtained by pointers to left prefix blocks of G. All the following $DIST$ columns are obtained by pointers to top prefix blocks of G.

Furthermore, the new column of DIST is computed in $O(t)$ time by using the 3 new edges in the new cell of G and the 3 $DIST$ columns computed for the top, diagonal and left prefix blocks of G as follows. Entry $[i]$ in column ℓ_c of the $DIST$ for G contains the weight of the optimal path from entry i in the input border of G to vertex (ℓ_r, ℓ_c). This path must go through one of the three vertices $(\ell_r - 1, \ell_c)$, $(\ell_r - 1, \ell_c - 1)$ or $(\ell_r, \ell_c - 1)$. Therefore, the weight of the optimal (highest scoring) path from entry i in the input border of G to (ℓ_r, ℓ_c) is equal to the maximum among the following three values:

1 Entry $[i]$ of column $\ell_c - 1$ of the $DIST$ for the left prefix of G, plus the weight of the horizontal edge leading into (ℓ_r, ℓ_c).

2 Entry $[i]$ of column $\ell_c - 1$ of the $DIST$ for the diagonal prefix of G, plus the weight of the diagonal edge leading into (ℓ_r, ℓ_c).

3 Entry $[i]$ of column ℓ_c of the $DIST$ for the top prefix of G, plus the weight of the vertical edge leading into (ℓ_r, ℓ_c).

In order to demonstrate how a prefix block can be accessed in constant time, we refer the reader to Figure 7. Here we have the tries for the phrases of A and B, as formed during the LZ78 parsing of

the strings. Each of the partitioned blocks in the alignment graph for comparing strings A and B corresponds to a unique node in the trie for A and a unique node in the trie for B. For example, block 5/4 corresponds to node 5 in the trie for A (phrase "ag") and to node 4 in the trie for B (phrase "acg"). The tries can be used to access direct prefix blocks in constant time. For example, to obtain the diagonal prefix block of G, we just have to backtrack up one edge in the trie for A (this leads to node 3), and up one edge in the trie for B (this leads to node 2), and this way we compute in constant time the id (3,2) of the diagonal prefix block of G. We refer the reader to [13] for the engineering details of the encoding algorithm.

The Alignment Stage

Given I and the $DIST$ for G which was computed in the encoding stage, the alignment stage can be implemented in $O(t)$ time, by using a variation of the alignment stage of Algorithm CSA1, extended to support full propagation from the leftmost and upper boundaries to the bottom and rightmost boundaries of G [13].

Time and Space Analysis of the CSA1/LZ78 Combination Algorithm.

Assuming sequence size n and sequence entropy $h \leq 1$. The LZ78 factorization algorithm parses the strings and constructs the tries for A and B in $O(n)$ time. The resulting number of phrases in both A and B is $O(hn/\log n)$. Since the work and space for each block (both encoding and propagation) is linear with the size of its I/O borders, the total time and space complexity is linear with the total size of the borders of the blocks. The block borders form $O(hn/\log n)$ rows of size $|B|$ each, and $O(hn/\log n)$ columns of size $|A|$ each, in the alignment graph. Therefore, the total time and space complexity is $O(hn^2/\log n)$.

4.3 Combining Algorithm CSA2 with LZ78 Block Partitioning.

The number of blocks in the LZ78-partitioned alignment graph is equal to the number of phrases in A times number of phrases in B, and is therefore $O(h^2n^2/(\log n)^2)$, assuming that both sequences share the same entropy h. When computing the optimal global alignment value with scoring matrices that follow the "discreteness" condition, algorithm CSA2 from section 3.2 can be extended to support full propagation from the leftmost and upper boundaries to the bottom and right most boundaries of G. This extended propagation algorithm can then be used to compute the values of the global alignment O for G, given the I for G and a minimal encoding of the $DIST$ for G. The advantage of this minimal encoding of $DIST$ is that rather than saving an $O(t)$ sized $DIST$ column per block, we only need to save a constant number of values per block. The encoding for the new $DIST$ column of each block can be computed and stored in constant time and space from the information stored for the left, diagonal and top prefix blocks of G, using the technique described in Section 6 of [59]. This reduces the space complexity to $O(h^2n^2/(\log n)^2)$, while preserving the $O(hn^2/\log n)$ time complexity.

4.4 Combining Algorithm CSA1 with Run Length Encoding Block Partitioning

In this solution, the alignment graph is also partitioned into blocks, where each block here consists of two runs – one of X and one of Y. This results in the partition of the alignment graph into $m'n'$ blocks (see Figure 6.) The suggested algorithm also propagates accumulated scores from the left and upper boundaries of each block, to its bottom and right boundaries.

The Encoding Stage

We claim that this algorithm does not require any encoding stage, as any value of the $DIST$ used for encoding a run-length block can be computed in constant time during the propagation stage, as follows. Consider the block R for comparing the run α^i of X with the run β^j of Y. An edge in R could be assigned one of three possible weight values: D(diagonal), H(horizontal) and V(vertical).

Let Δ_h and Δ_w denote the difference in row index values and column index values respectively, between entry i on the input border of R, and entry j on the output border of R.

We show how to compute $DIST[i, j]$ (which is the cost of the best scoring path from entry i in the input border of the block, to entry j in the output border of the block) in constant time, given Δ_h and Δ_w for the input and output entries, and the values D, H and V.

- $H + V \leq D$. Clearly, an optimal (highest scoring) path from i to j can use all possible diagonal edges and only then the minimal number of remaining H and V edges necessary to reach j.

  Therefore, $DIST[i, j]$ obtains one of three values:

  1. If $\Delta_w = \Delta_h$, then $DIST[i, j] = D \times \Delta_h$.
  2. If $\Delta_w > \Delta_h$, then $DIST[i, j] = D \times \Delta_h + H \times (\Delta_w - \Delta_h)$.
  3. If $\Delta_w < \Delta_h$, then $DIST[i, j] = D \times \Delta_w + V \times (\Delta_h - \Delta_w)$.

- $H + V > D$. In this case, an optimal path never uses any diagonal edge. The path includes only the minimal number of H edges, and the minimal number of V edges necessary to reach j from i. In this case, $DIST[i, j] = H \times \Delta_w + V \times \Delta_h$.

Therefore, $DIST[i, j]$ can be easily computed in constant time for any additive-gap scoring scheme which uses a scoring matrix of real numbers.

The Alignment Stage

Given I and the $DIST$ for G which was computed in the encoding stage, the alignment stage can be implemented for each block in time linear with the size of its I/O borders, by using a variation of the alignment stage of Algorithm CSA1, extended to support full propagation from the leftmost and upper boundaries to the bottom and rightmost boundaries of R [13].

Time and Space Analysis of the CSA1/LZ78 Combination.

The O vector for each block is computed using *SMAWK*. Vector I for block R can be easily obtained from the O vectors for the block above R and the block to its left, in time linear with the sides of R. Therefore, any value $OUT[i, j] = I[i] + DIST[i, j]$ can be computed in constant time.

Since the work for each block is linear with the size of its I/O borders, the total time complexity is linear with the total size of the borders of the blocks, which is $O(m'n + n'm)$.

Since all the relevant $DIST$ entry values are computed "on the fly" and do not need to be stored, Hirshberg's method [29] can be applied to achieve an algorithm with a space complexity which is linear with the size of the uncompressed strings.

Acknowledgements.

Many thanks to Amihood Amir, Alberto Apostolico and Ilan Newman for their helpful comments.

References

[1] Amir, A., G. Benson, and M. Farach, Let sleeping files lie: Pattern matching in Z-compressed files. *J. of Comp. and Sys. Sciences*, **52**(2), 299–307 (1996).

[2] A. Apostolico, String editing and longest common subsequences. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, **2**, 361–398, Berlin, 1997. Springer Verlag.

[3] Apostolico, A., M. Atallah, S. E. Hambrusch, New Clique and Independent Set Algorithms for Circle Graphs. *Discrete Applied Mathematics.*, **36**, 1–24 (1992).

[4] Apostolico, A., M. Atallah, L. Larmore, and S. McFaddin, Efficient parallel algorithms for string editing problems. *SIAM J. Comput.*, **19**, 968–998 (1990).

[5] Apostolico, A., G.M. Landau and S. Skiena, Matching for Run Length Encoded Strings, *Journal of Complexity*, **15**(1), 4–16 (1999).

[6] Aggarwal, A., M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric Applications of a Matrix-Searching Algorithm, *Algorithmica*, **2**, 195–208 (1987).

[7] Arbell, O., G. M. Landau, and J. Mitchell, Edit distance of run-length encoded strings, *Information Processing Letters*, **83**(6), 307–314 (2002)

[8] Aggarawal, A., and J. Park, Notes on Searching in Multidimensional Monotone Arrays, *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 497–512 (1988).

[9] Bell, T.C., J.C. Cleary, and I.H. Witten. *Text Compression.* Prentice Hall, (1990).

[10] Benson, G., A space efficient algorithm for finding the best nonoverlapping alignment score, *Theoretical Computer Science*, **145**, 357–369 (1995).

[11] Bunke, H., and J. Csirik. An improved algorithm for computing the edit distance of run length coded strings, *Information Processing Letters*, **54**, 93–96 (1995).

[12] Chao, K. M., R. Hardison, and W. Miller, Recent developments in linear-space alignment methods: a mini survey. *J. Comp. Biol.*, **1**, 271–291 (1994).

[13] Crochemore, M., G.M. Landau and M.Ziv-Ukelson, "A Sub-quadratic Sequence Alignment Algorithm for Generalized Cost Metrics" *SIAM Journal of Computing*, **32**(6), 1654–1673 (2003).

[14] Crochemore, M., Transducers and Repetitions. *Theoret. Comput. Sci.*, **45**, 63–86 (1986).

[15] Crochemore, M., and W. Rytter, Text Algorithms, *Oxford University Press*, (1994).

[16] Eppstein, D., Sequence Comparison with Mixed Convex and Concave Costs, *Journal of Algorithms*, **11**, 85–101 (1990).

[17] Elias, P., Universal Codeword Sets and Representation of Integers, *I.E.E.E. Transf. Inform. Theory*, **IT21**(2), 194–203 (1975).

[18] Eppstein, D., Z. Galil, and R. Giancarlo, Speeding Up Dynamic Programming, *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 488–296 (1988).

[19] Eppstein, D., Z. Galil, R. Giancarlo, and G.F. Italiano, Sparse Dynamic Programming I: Linear Cost Functions, *JACM*, **39**, 546–567 (1992).

[20] Eppstein, D., Z. Galil, R. Giancarlo, and G.F. Italiano, Sparse Dynamic Programming II: Convex and Concave Cost Functions, *JACM*, **39**, 568–599 (1992).

[21] Farach, M., and M. Thorup, String matching in Lempel-Ziv compressed strings. *Algorithmica*, **20**, 388–404 (1998).

[22] Giancarlo, R., Dynamic Programming: Special Cases, *Pattern Matching Algorithms*, edited by Apostolico, A., and Z. Galil, Oxford University Press, 201–232 (1997).

[23] Gabow, H.N., and R.E. Tarjan, A Linear Time Algorithm for a Special Case of Disjoint Set Union. *J. Comput. Syst. Sci*, **30**, 209–221 (1985).

[24] Galil, Z., and R. Giancarlo, Speeding Up Dynamic Programming with Applications to Molecular Biology, *Theoretical Computer Science*, **64**, 107–118 (1989).

[25] Galil Z., and K. Park, A linear-time algorithm for concave one-dimensional dynamic programming, *Info. Processing Letters*, **33**, 309–311 (1990).

[26] Gasieniec, L., M. Karpinski, W. Plandowski, W. Rytter, Randomised efficient algorithms for compressed strings: the finger-print approach, *Proc. 7th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1075, 39–49 (1996).

[27] Gasieniec, L., and W. Rytter, Almost optimal fully LZW compressed pattern matching, *Data Compression Conference*, J. Storer, ed, (1999).

[28] Gusfield, D., Algorithms on Strings, Trees, and Sequences. *Cambridge University Press*, (1997).

[29] Hirschberg, D.S., A Linear Space Algorithm for Computing Maximal Common Subsequences, *Communications of the ACM*, **18**(6), 341–343 (1975).

[30] Hirshberg, D.S., "Algorithms for the longest common subsequence problem", *JACM*, **24**(4), 664–675 (1977).

[31] Hirshberg, D.S., and L.L. Larmore, The Least Weight Subsequence Problem, *SIAM J. Compt.*, **16**(4), 628–638 (1987).

[32] Huang, X., and W. Miller, A time-efficient, linear space local similarity algorithm, *Adv. Appl. Math.*, **12**, 337–357 (1991).

[33] Kannan, S.K., and E.W. Myers, An Algorithm For Locating Non-Overlapping Regions of Maximum Alignment Score, *SIAM J. Comput.*, **25**(3), 648–662 (1996).

[34] Karkkainen, J., G. Navarro, and E. Ukkonen, Approximate String Matching over Ziv-Lempel Compressed Text, *Proc. 11th Annual Symposium On Combinatorial Pattern Matching* 195–209 (2000).

[35] Karkkainen, J., and E. Ukkonen, Lempel-Ziv parsing and sublinear-size index structures for string matching, *Proc. Third South American Workshop on String Processing (WSP '96)*, 141–155 (1996).

[36] Kida, T., M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa, Shift-And approach to pattern matching in LZW compressed text, *Proc. 10th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1645, 1–13 (1999).

[37] Klawe, M., and D. Kleitman, An Almost Linear Algorithm for Generalized Matrix Searching, *SIAM Jour. Descrete Math.*, **3**, 81–97 (1990).

[38] Landau, G.M., E.W. Myers, and J.P. Schmidt, Incremental String Comparison, *SIAM J. Comput.*, **27**(2), 557–582 (1998).

[39] Landau, G.M., E.W. Myers, and M. Ziv-Ukelson, Two Algorithms for LCS Consecutive Suffix Alignment, *Proc. 15th Annual Symposium On Combinatorial Pattern Matching*, LNCS 3109, 173–193 (2004).

[40] Landau, G.M., and M. Ziv-Ukelson, On the Common Substring Alignment Problem, *Journal of Algorithms*, **41**(2), 338–359 (2001)

[41] Landau, G.M., B. Schieber, and M.Ziv-Ukelson, "Sparse LCS Common Substring Alignment" *Information Processing Letters*, **88**(6), 259–270 (2003).

[42] Lempel, A., and J. Ziv, On the complexity of finite sequences, *IEEE Transactions on Information Theory*, **22**, 75–81 (1976).

[43] Levenshtein, V.I., Binary Codes Capable of Correcting Deletions, Insertions and Reversals, *Soviet Phys. Dokl*, **10**, 707–710 (1966).

[44] Ziv, J., and A. Lempel, A Universal Algorithm for Sequential Data Compression, *IEEE Transactions on Information Theory*, **IT-23**(3), 337–343 (1977).

[45] Ziv, J., and A. Lempel, Compression of individual sequences via variable rate coding, *IEEE Trans. Inform. Th.*, **24**, 530–536 (1978).

[46] Mitchell, J., A Geometric Shortest Path Problem, with Application to Computing a Longest Common Subsequence in Run-Length Encoded Strings, Technical Report, Dept. of Applied Mathematics, SUNY Stony Brook, 1997.

[47] Main, M. G., R. J. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, **5**, 422–432 (1984).

[48] Navarro G., T. Kida, M. Takeda, A. Shinohara, and S. Arikawa: Faster Approximate String Matching Over Compressed Text, *Proc. Data Compression Conference (DCC2001)*, IEEE Computer Society, 459–468 (2001).

[49] Mäkinen, V., G. Navarro, and E. Ukkonen, Approximate Matching of Run-Length Compressed Strings, *Proc. 12th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1645, 1–13 (1999).

[50] Mäkinen, V., G. Navarro, and E. Ukkonen, Approximate Matching of Run-Length Compressed Strings, *Algorithmica*, **35**, 347–369 (2003).

[51] Manber, U., A text compression scheme that allows fast searching directly in the compressed file, *Proc. 5th Annual Symposium On Combinatorial Pattern Matching*, LNCS 2089, 31–49 (2001).

[52] Masek, W.J., and M.S. Paterson, A faster algorithm for computing string edit distances. *J. Comput. Syst. Sci.*, **20**, 18–31 (1980).

[53] Miller, W., and E.W. Myers, Sequence Comparison with Concave Weighting Functions, *Bull. Math. Biol.*, **50**, 97–120 (1988).

[54] Monge, G., Deblai et Remblai, *Memoires de l'Academie des Sciences*, Paris (1781).

[55] Navarro, G., and M. Raffinot, A general practical approach to pattern matching over Ziv-Lempel compressed text, *Proc. 10th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1645, 14–36 (1999).

[56] Navarro, G., and M. Raffinot. Boyer-Moore string matching over Ziv-Lempel compressed text, *Proc. 11th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1848, 166–180 (2000).

[57] Needleman, S.B., and C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of of two proteins, *J. Mol. Biol.*, **48**, 443–453 (1970)

[58] Sankoff D., and J.B. Kruskal (editors), *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, (1983).

[59] Schmidt, J.P., All Highest Scoring Paths In Weighted Grid Graphs and Their Application To Finding All Approximate Repeats In Strings, *SIAM J. Comput*, **27**(4), 972–992 (1998).

[60] Shabita Y., T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, S. Arikawa, *Speeding up pattern matching by text compression*, *CIAC 2000*, LNCS 1767, 306–315 (2000).

[61] Smith, T. F. and M. S. Waterman, Identification of common molecular subsequences, *J. Molecular Biol.*, **147**, 195–197 (1981).

[62] Ukkonen, E., Finding Approximate Patterns in Strings, *J. Algorithms*, **6**, 132–137 (1985).

[63] Welch, T.A., A Technique for High Performance Data Compression, *IEEE Trans. on Computers*, **17**(6), 8–19 (1984).

An Appendix Including All Figures

T	=	"BCBADBDCD"	Y	=	*"BCBD"*			
S_1	=	"BC *BCBD* C"	B_1	=	"BC"	F_1	=	"C"
S_2	=	"E *BCBD* D*BCBD* A"	B_{2a}	=	"E"	F_{2a}	=	"D*BCBD* A"
			B_{2b}	=	"E*BCBD* D"	F_{2b}	=	"A"

Figure 1: An example of two different source strings S_1, S_2 sharing a common substring Y, and a target string T.

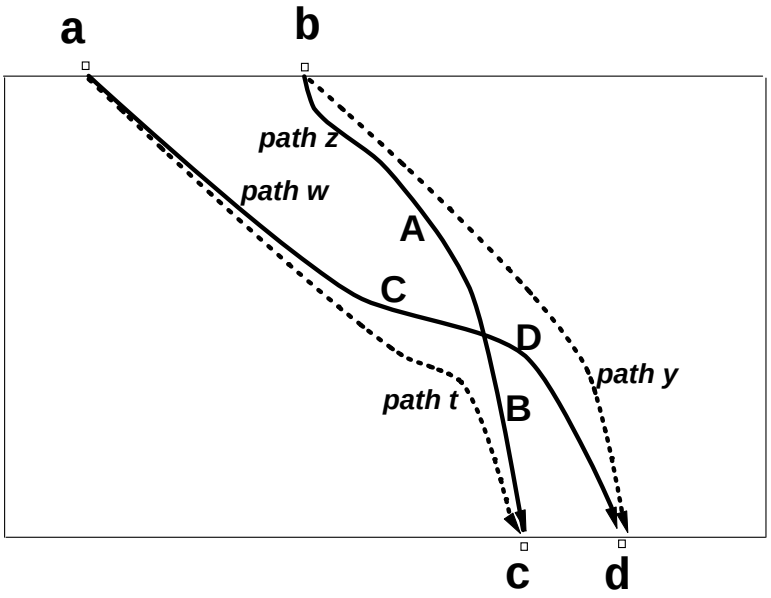

Figure 2: Optimal paths that must cross.

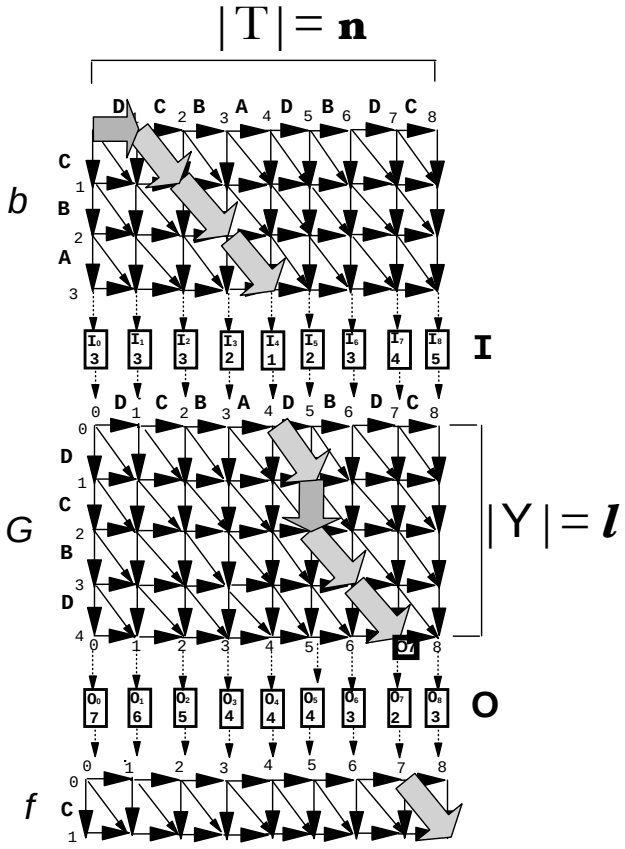

Figure 3: The Dynamic Programming graph for computing the distance between $T =$ "$DCBADBDC$", and $S_2 =$ "$CBADCBDC$". S_2 contains the common substring $Y =$ "$DCBD$". This figure continues the example of Figure 1.

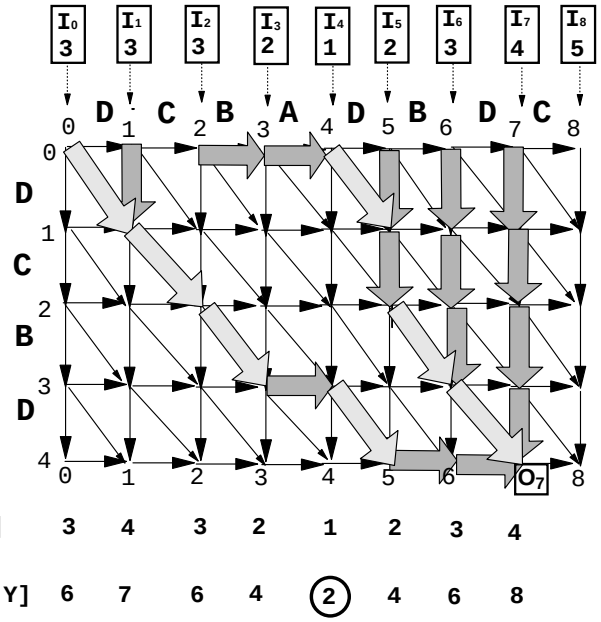

Figure 4: The computation of output entry O_7 for $T =$ "$DCBADBDC$", $B_i =$ "CBA", and $Y =$ "$DCBD$". Let T_u^z, such that $1 \leq u \leq z \leq n$, denote the substring of T from index u (inclusive) up to index z (inclusive). The minimal output at O_7, $\min_{x=0}^{7}\{I_x + Edit[T_{x+1}^7, Y]\} = 2$, is achieved by the path originating at column 4 and receiving input $I_4 = 1$. This figure continues the example of Figures 1 and 3.

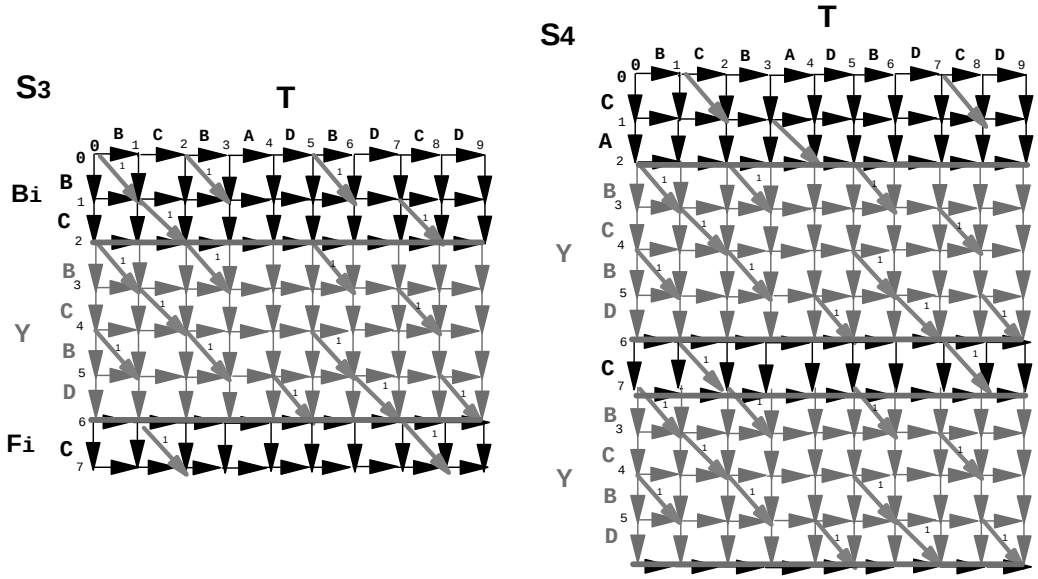

Figure 5: Examples of two dynamic programming graphs for the alignment of two source sequences $S_3 =$ "$BCBCBD$" and $S_4 =$ "$CABCBDCBCBD$", sharing the common substring $Y =$ "$BCBD$", with target string $T =$ "$BCBADBDCD$". Note that Y is repeated twice in S_4. The target sequence is $T =$ "$BCBADBDCD$".

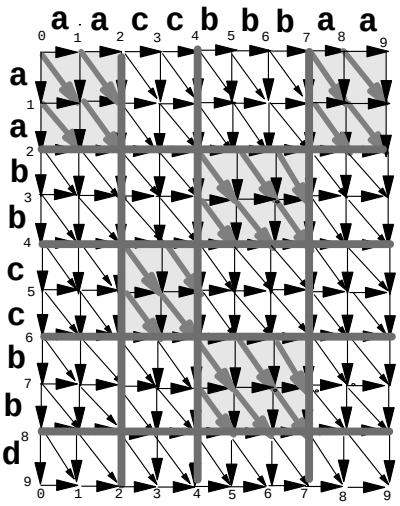

Figure 6: The block partitioned alignment graph obtained by applying run-length encoding to the two compared strings $A = aabbccbbd = a^2b^2c^2b^2d^1$ and $B = aaaccbbbaa = a^3c^2b^3a^2$.

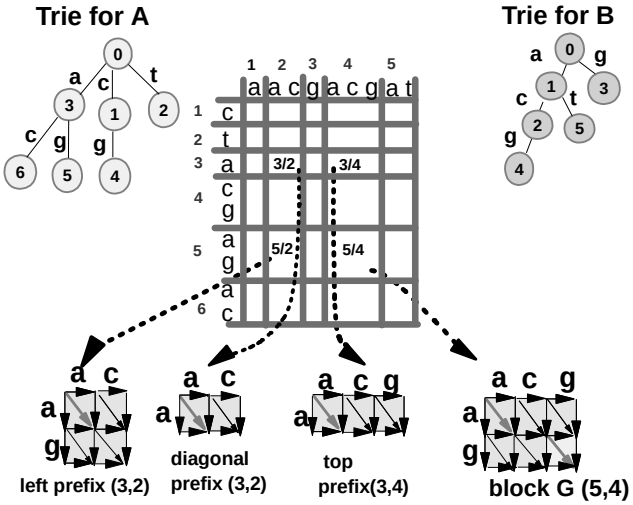

Figure 7: The block partition of the alignment graph, and the corresponding tries, obtained by LZ78 parsing of strings $A =$ "*ctacgaga*" and $B =$ "*aacgacga*". Note that the structure of all cells but the one at the rightmost, lowest corner of the alignment graph have appeared in previous blocks. .

Column Indices

	0	1	2	3	4	5	6	7	8
	3	3	3	2	1	2	3	4	5

Input Row I

DIST[x,j] = Edit[T^{j}_{x+1},Y]

0	4	3	2	1	1	1	2	3	4
1	–	4	3	2	2	2	3	4	5
2	–	–	4	3	3	3	4	3	4
3	–	–	–	4	4	3	3	2	3
4	–	–	–	–	4	3	2	1	2
5	–	–	–	–	–	4	3	2	3
6	–	–	–	–	–	–	4	3	2
7	–	–	–	–	–	–	–	4	3
8	–	–	–	–	–	–	–	–	4

OUT[x,j] = I_x + DIST[x,j]

0	7	6	5	4	4	4	5	6	7
1	–	7	6	5	5	5	6	7	8
2	–	–	7	6	6	6	7	6	7
3	–	–	–	6	6	5	5	4	5
4	–	–	–	–	5	4	3	2	3
5	–	–	–	–	–	6	5	4	5
6	–	–	–	–	–	–	7	6	5
7	–	–	–	–	–	–	–	8	7
8	–	–	–	–	–	–	–	–	9

Output Row O

7	6	5	4	4	4	3	2	3

Figure 8: The *DIST* and *OUT* matrices which correspond to the sequences $T =$ "*DCBADBDC*", $Y =$ "*DCBD*" and the input row I for $B_i =$ "*CBA*". This figure continues the example of Figures 3 and 4.

OUT Table

	0	1	2	3	4	5	6	7	8
0	7	6	5	4	4	4	5	6	7
1	–	7	6	5	5	5	6	7	8
2	–	–	7	6	6	6	7	6	7
3	–	–	–	6	6	5	5	4	5
4	–	–	–	–	5	4	3	2	3
5	–	–	–	–	–	6	5	4	5
6	–	–	–	–	–	–	7	6	5
7	–	–	–	–	–	–	–	8	7
8	–	–	–	–	–	–	–	–	9

DELTA[i,j]=OUT[i,j]-OUT[i,j-1]

=DIST[i,j]-DIST[i,j-1]

0	~	-1	-1	-1	0	0	1	1	1
1	~	~	-1	-1	0	0	1	1	1
2	~	~	~	-1	0	0	1	-1	1
3	~	~	~	~	0	-1	0	-1	1
4	~	~	~	~	~	-1	-1	-1	1
5	~	~	~	~	~	~	-1	-1	1
6	~	~	~	~	~	~	~	-1	-1
7	~	~	~	~	~	~	~	~	-1

Borderline Points- O(ψ n)

–	–	–	–	–	2	2	1	5
–	–	–	–	–	–	3	1	5

Figure 9: An example of a *DELTA* matrix and its Borderline Points. Note that for the Edit Distance metric, which is used in this example, $\psi = 2$, and therefore the number of Borderline Points for *DELTA* is bounded by $2n$. This figure continues the example of Figures 3, 4 and 8.

OUT Table:

	0	1	2	3	4	5	6	7	8
0	(7)	(6)	(5)	(4)	(4)	4	5	6	7
1	–	7	6	5	5	5	6	7	8
2	–	–	7	6	6	6	7	6	7
3	–	–	–	6	6	5	5	4	5
4	–	–	–	–	5	(4)	(3)	(2)	(3)
5	–	–	–	–	–	6	5	4	5
6	–	–	–	–	–	–	7	6	5
7	–	–	–	–	–	–	–	8	7
8	–	–	–	–	–	–	–	–	8

Input Row I

3	3	3	2	1	2	3	4	5

Borderline Points

–	–	–	–	–	2	2	1	5
–	–	–	–	–	–	3	1	5

Candidate List(row index)

0	0	0	0	0	4	4	4	4
	1	1	1	4	5	5	5	6
		2	3			6	6	7
							7	8

Output Row O

7	6	5	4	4	4	3	2	3

Figure 10: A trace of the contents of the candidate list, during iterations 0 to 8 of the Alignment Stage Algorithm. The lightly shaded entries of *OUT* are values which have become extinct. This figure continues the example of Figures 3, 4, 8 and 9.

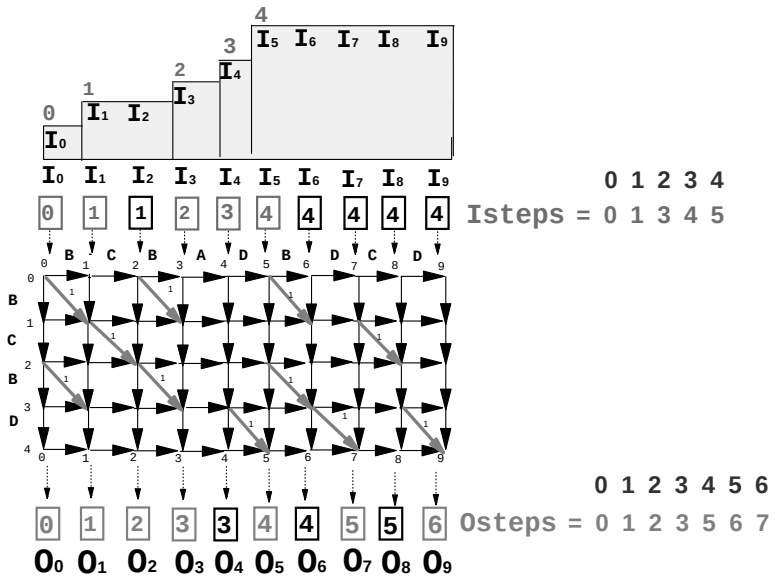

Figure 11: Due to the unit-step properties of LCS, both I and O are monotonically non-decreasing series, and their values go up by unit steps. Therefore, the compressed $Isteps$ and $Osteps$ vectors are constructed which store only the steps of I and O, correspondingly.

$DIST[x,j] = LCS[T^{j}_{x+1}, Y]$

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	3	4	4	4	4	4
1	–	0	1	2	2	3	3	3	3	3
2	–	–	0	1	1	2	2	3	3	3
3	–	–	–	0	0	1	1	2	2	3
4	–	–	–	–	0	1	1	2	2	3
5	–	–	–	–	–	0	1	2	2	3
6	–	–	–	–	–	–	0	1	1	2
7	–	–	–	–	–	–	–	0	1	2
8	–	–	–	–	–	–	–	–	0	1
9	–	–	–	–	–	–	–	–	–	0

DIST Compressed by its steps:

	0	1	2	3	4	k = 1..Ly
0	0	1	2	3	5	
1	1	2	3	5	-	
2	2	3	5	7	-	
3	3	5	7	9	-	
4	4	5	7	9	-	
5	5	6	7	9	-	
6	6	7	9	-	-	
7	7	8	9	-	-	
8	8	9	-	-	-	
9	9	-	-	-	-	

Figure 12: Each row of the LCS $DIST$ is also a monotonically non-decreasing series. Therefore, the LCS $DIST$ can be compressed into an $O(nL)$-sized table which stores only the steps of the rows of $DIST$.

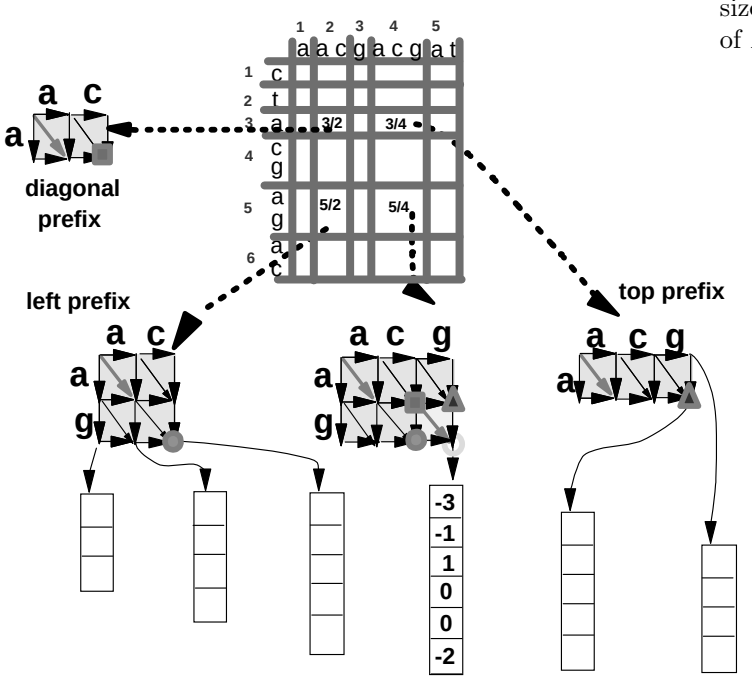

Figure 13: [13] Computing a single entry in the new $DIST$ column for the new cell of G in constant time. This figure continues Figures 7 and 14.

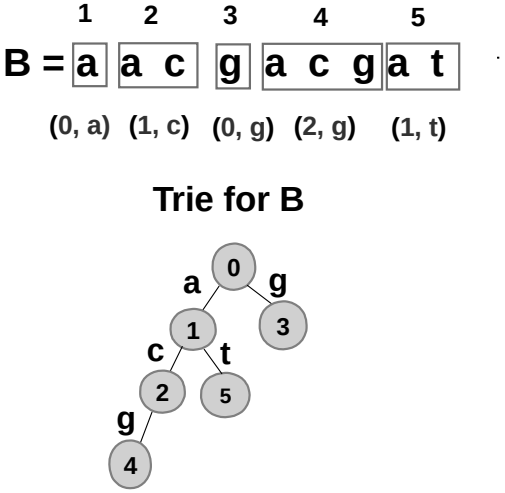

Figure 14: A demonstration of LZ78 parsing on text "$B = aacgacgat$", and the corresponding dictionary trie.